2
Concepts and Technologies

In [Boehm and Basili, 2001] it is stated that about 50% of the deployed software contains non-trivial defects. Although their statement is directed towards user developed software (e.g. spread-sheets), it is frequently mentioned that software in general is still too failure prone. As mentioned in the PITAC[1] Report to the President of the USA (1999), section 2 *Priorities for Research*:

> The demand for software has grown far faster than our ability to produce it. Furthermore, the Nation needs software that is far more usable, reliable, and powerful than what is being produced today. We have become dangerously dependent on large software systems whose behavior is not well understood and which often fail in unpredicted ways. Therefore, increases in research on software should be given a high priority. Special emphasis should be placed on developing software for managing large amounts of information, for making computers easier to use, for making software easier to create and maintain, and for improving the ways humans interact with computers.

It is our hypothesis that developing recovery oriented software is an effective step to reduce the failure proneness of software. Much work has already been done in this area, as we will show in the next section. Most of this effort is focused on web-based systems. We are also interested in non web-based software as well as in embedded software.

2.1.
Recovery Oriented Software development process characteristics

As stated in the previous chapter, it is not the goal of this work to propose the modification of an existing, or even to propose a new software development process. However, we are fully convinced, by our experiments, that a recovery oriented software development process must concentrate efforts in the following areas:

---

[1] PITAC President's Information Technology Advisory Committee (1999) - Report to the President of the USA. URL: http://www.nitrd.gov/pitac/report/exec_summary.html

2.1.1. Defect prevention effort

This is the effort spent during development time to avoid the presence of defects in a system. It concentrates mainly on specification, architecture, design, coding, inspection, test, verification and validation. Essentially, it corresponds to the conventional development effort, as well as to the maintenance or evolution[2] effort once the cause of a problem has been identified. Obviously, this effort can be reduced by means of the use of best practices aiming at correctness by development[3], maintainability and evolveability.

2.1.2. Potential failure detection effort

It is important to distinguish between two distinct types of defects: the ones that are under the control of the developer and that we wish to minimize by means of prevention, and the ones that are not under the developer's control, whose causes are usually external to the system, such as hardware failure or interference from another system. These latter failures require a specific design effort, enabling the corresponding errors to be observed and properly handled.

The potential failure detection effort is the effort spent during development time to identify failures that might happen at runtime. Such failures can be caused by inaccurate coding, but may also be caused externally to the software due to exogenous errors. The major characteristic of potential failures is that they are not known beforehand. This effort includes not only the design overhead, but also operational costs, i.e. the computational effort spent in control actions that do not directly contribute to software functionalities. It is represented by artifacts like dedicated hardware and software, use of redundancy or even software clones [Staa, 2000] as well as replicas in order to detect inconsistencies by comparing

---

[2] We distinguish between ***maintenance*** (corrective, perfective and preventive maintenance) and ***evolution*** (enhancement and adaptive maintenance) [Kemerer and Slaughter, 1997]

[3] Software is said to be ***correct by construction*** if it contains no defect previous to the first test. It is said to be ***correct by development*** if it contains no defect previous to distribution or deployment.

different outputs from a single input [Pullum, 2001], like some techniques used in fault tolerance. As examples of software dedicated to failure detection, we can mention:

1. Self-checkable data structures: these are data-structures containing redundancy that allows verifying their structural consistency (e.g. conformance with structural invariant assertions) without requiring knowledge of the system state [Taylor, 1986];
2. Data-structure verifiers: code designed to verify self-checkable data structures [Staa, 2000];
3. Data-structure recoverers: code designed to recover damaged data-structures. When such code is embedded within the data-structure, it is considered a "recoverable data-structure".
4. Self-test functions: created to run self-consistence tests in a system.

It is possible to conclude three things:

1. A recovery oriented software has components dedicated exclusively to failure detection, as well as having an internal organization suitable to allow for data verification during runtime; and
2. Software must be designed to be recovery oriented. Such a feature is very expensive and error-prone to be added to already developed software.
3. But recovery oriented software must provide more things – once a failure has been detected, some action must be triggered recovering the system to a state from which it may continue working dependably. Furthermore, this action must require a very small amount of time (few minutes, or sometimes even fractions of minutes). We will discuss this subject in more detail in the chapter Recovery Techniques

2.1.3. Failure handling effort (FHE)

This is the effort spent during development and maintenance time (architecture, design, implementation and test) to add ways to handle failures

detected during runtime (see item above). By failure handling we mean “recover as gracefully as possible” (although there is no clear way to measure “how gracefully a failure has been handled”), i.e.:

- minimizing the need of manual intervention
- reducing the loss of work;
- giving precise information to the user about what happened, using his/her viewpoint, and how to continue working in the best possible way.

For example, a failure detected in a sensor or in the software that controls the sensor can be handled automatically, maybe by removing the defective sensor from operation and starting a new equivalent one, or simply deactivating the defective sensor and leaving the task to the remaining ones - although the system will operate in a degraded way, the reliability of other operations possibly will not be affected. In other cases, the best thing to do is to roll back to a previous safe state and restart the program in order to minimize damage propagation.

In addition to handling failures as they are detected, it is also necessary to remove, or, in case of external failures, encapsulate them, which means isolating the system from them – for example, if a magnetic sensor must acquire data between 1 and 10 Gauss, but is known for giving sometimes spurious readings, then the system must be aware of that and, hence, identify and ignore spurious values maintaining the amount of false positives under control.. Considerable effort is then spent in order to provide means to either encapsulate or eliminate the causes of a failure.

For web applications, different from conventional fault tolerance, failure recovery may require some user intervention. The user can provide valuable information to minimize loss of data or may even require that no recovery be made – in such cases the user will attempt to recover the data himself. For example, if the system detects an invalid record in a database, it is important to ask the user before whether it should either simply erase it, or mark it as bad – which would be preferred for debugging purposes, but implies that the design must have taken the “marked bad” state into consideration. In other cases, the system must decide if the failure is severe enough to require its termination, or if a degraded operation is acceptable [Bentley, 2005]. Essentially, the goal is to keep the system operating, even if degraded, while the defective component is being repaired. According to [Fox and Patterson, 2002], there exist specific MTTR

thresholds below which the user experience is not appreciably disrupted by a transient failure (so that further improvement beyond that threshold yields only marginal benefit) and another above which it is intolerable (so that users give up or click over to a competitor). In other words, users tolerate failures to some extent. The question is how to keep the system operating dependably within this extent.

For embedded or control systems, user intervention is almost always impossible. The system must decide for itself what to do in order to keep running. This means that, if a failure is detected, it should be properly handled – if internal, the components that caused it should be deactivated and, if external, it should be encapsulated - which means that, every time the external condition that triggers the failure is detected, the system should act in order to avoid significant consequences of this failure.

The effort spent in this item corresponds typically to designing, implementing and testing recovery code, whose purpose is:

1. when detecting a failure, to restore the system to a valid state; or
2. perform a "house cleaning" procedure (for example, code that terminates the system in order to preserve or even restore data integrity of persistent data; or code to roll-back the system to a previous safe state); or
3. provide redundancy of hardware or software in order to guarantee service availability.

2.1.4. Defect removal effort (DRE)

This is the effort spent to both identify and eliminate defects, i.e. the causes responsible for the failures, or to encapsulate failures in such a way that potential damages are certainly kept below an acceptable level. The problem here is that such defects have not been identified when testing or using software. Instead we observe failures caused by these defects and, according to the symptoms associated with the failures, one tries to find the offending defect. As mentioned before, failures may be due to defects in the code, but may also be due to external factors. In the latter case, it is necessary to include code that detects malfunction.

For example, in case of data entry, it is strongly recommended to add data verification code. In the case of gathering data from sensors, each datum could be compared to a valid range or to the current mean of the read values, in order to detect potential outliers. Once identified and isolated, defects can be removed or failures could be encapsulated.

As mentioned before, according to Basili and Boehm [Basili and Boehm, 2001], more than 50% of software systems contain non-trivial defects. For these non-trivial defects, the failure analysis effort may be very hard or impossible to estimate. One way to reduce this effort is to invest resources in failure detection (item 2), as early detection not only contributes to avoiding inconsistent states and data from spreading throughout the system, potentially increasing damage, but allied to a debugging data collection policy may also help to identify and isolate the corresponding defect. Another important issue is that, the faster one detects a failure, the closer (considering time) the detecting code will be to the point where the error occurred, thus reducing the FDE.

Once the responsible defect is detected, it is necessary to decide if it will be removed – in some cases the defect removal may be too costly compared to the impact caused on the service provided by the system. For example this would be the case when the DRE is too expensive. This is the case when a defect is found in embedded software that is installed in many equipments. The costs of an upgrade may prohibitive. Another case is for software that is used in critical processes that cannot be stopped to allow redeployment without impacting its production.

Even though a great part of the DRE takes place at production time, it is during the development time that instruments must be developed so that the mean time to identify the cause of a failure, to remove it and to deploy the corrected version is assuredly kept below a given level. Such instruments are mainly pieces of code that gather as much *relevant* information about the system state as possible, thus facilitating defect identification.

2.2. Fault tolerant software versus recovery oriented software

Fault tolerance is the property that enables a software system to continue operating properly in the event of a failure of some of its components [Wikipedia,

2008]. This is directly related to failure handling, and it means that fault tolerance is a characteristic of recovery oriented software. However, recovery oriented software must care about some other issues:

Recovery oriented software must be able to provide accurate information that help the developers fix it when a failure is reported, even during development time [Candea and Fox, 2003] when in the hands of the testers;

Recovery oriented software must provide means to quickly restore the workbench and minimize possible damages to the service provided as well as to the execution environment when a failure is detected [Fox and Patterson, 2002];

Recovery oriented software must be able to re-establish full operational power as fast as possible (this means that operating in a degraded way cannot be simply accepted as a possible long-term solution for failure handling) [Fox and Patterson, 2002], [Candea and Fox, 2003];

2.3. Technologies and tools that support the development of recovery oriented software

There are many technologies, tools and best practices that enable the development of recovery oriented software. Some of them are listed as follows:

2.3.1. Debuggable Software

Debuggable software is written containing dedicated instruments (pieces of code) to provide specific and accurate information for the developers in case of failure, thus helping the fault removal process. One of the consequences of the development of debuggable software is the reduction of the number of non-trivial faults [Boehm and Basili, 2001] that are identified during the production time of the software. In order to understand the reason for this statement, it is necessary to analyze what makes a fault non-trivial. We start by considering the fault removal effort (DRE), whether the faults are trivial or not. We split DRE into two parts:

1) *Fault diagnose effort (FDE):*

We do not observe faults but rather their symptoms, i.e. failures. This is also true for the debugging process. Using fault diagnosis applied to a given failure, we search and determine its corresponding fault.

2) *Fault Correction Effort (FCE):*

After a failure has been diagnosed, the corresponding defect must be fixed or encapsulated. In addition, we must demonstrate that it has been correctly and completely dealt with and that the new deployable version of the software contains the complete correction. It is important to state that not every defect can be removed. For example, external errors such as data transmission errors occur due to unpredictable and inevitable causes. However, often it is possible to add redundancy (e.g. sum checks), allowing the detection of the error and, subsequently, to control its possible damage. Inserting the redundancy, the corresponding detection code and the damage handling code is what we call *error encapsulation.*

One of the major problems with defect removal is to estimate, a priori, the FDE. Even though the FCE might be considerable, once the fault has been diagnosed, removing it corresponds to conventional software development or maintenance activities. Hence there is already a large amount of knowledge and experience regarding FCE. According to [Satpathy et al, 2004], FCE can be reduced using best development practices.

Nevertheless, very little effort is being spent developing techniques designed to reduce the FDE, which is still very dependent on programmer skills and on sophisticated debugging environments. One way of reducing the FDE could be achieved by reducing or ideally eliminating non-trivial faults. However, as mentioned before, this seems to be a utopian proposal. Hence, an alternative would be to transform non-trivial into trivial faults. This leads to our definition of debuggable software:

Debuggable software is one in which the chances of observing a non-trivial fault is very low. Aside from assuring that it is easy to correct the software after some failure, debuggable software also promotes the development of software containing very few defects (faults). Hence the chance for failing should be rather small when considering debuggable software.

In other words, debuggable software is explicitly developed to reduce the number of non-trivial faults and, consequently, contributes to reduce the Fault Diagnose Effort as well. Debuggable software fits the principles of recovery oriented software – the more debuggable it is, the more recovery oriented it is.

One problem that could be stated is how could one assure that non-trivial faults do not exist. A less ambitious goal could be similar to fault-based testing [Morell, 1990], where the absence of a set of known classes of faults is verified by means of specific tests. Hence a less ambitious definition would be: Debuggable software is one in which the chances of observing a non-trivial fault of a given category is very low, approaching zero. The problem with this approach is that we are bound to a set of known fault categories, which, although possibly increasing as time passes, does not assure the absence of non-trivial errors. This approach is quite similar to fault-based testing [Morell, 1990]. It is beyond the scope of this work to detail this issue.

How could one reduce the number of non-trivial errors (internal or external) without prior knowledge of the error? The key idea is to prepare the software for the *error instant*. The error instant is the very instant when an error occurs. When an internal defect is exercised, or when an external error occurs, an error may be generated. When this error is observed, we have a failure. As long as it is not observed, it is still an error, not a failure. Thus, we cannot diagnose errors, but could possibly diagnose failures. The longer the time passed from the instant the error is generated to the instant of its observation, the harder will be the failure analysis and, consequently, the greater will be the FDE. Furthermore, the damage provoked by the malfunction might increase considerably. The error instant is thus one of the most important, if not the most important, event considering the fault removal effort. It is the ideal instant to collect information that will help to identify the related faults. When debuggable software fails, it must be able to provide precise information about itself:

- Information about internal state of execution: for example, allocated memory, variables, objects, threads, allocated resources, etc;
- Information about the environment state of execution: for example, database state, established socket connections, execution logs, etc;

- Information about how to reproduce the failure: for example, method invocations stack, user interactions with the interface, etc. In non-deterministic software (e.g., multi-threading or distributed systems), exact failure reproduction may be very hard or even impossible to be achieved. Debuggable software should help the developer to identify and isolate the causing thread, providing means to reproduce exactly the failure.

2.3.2. Software Components

Software components are widely present in current literature as a key technology for developing recovery oriented software [Fox and Patterson, 2002], [Candea and Fox, 2003], [Candea et al, 2003], [Fox, 2002], [Aghdaie and Tamir, 2003], [Castro et al 2003], [Ponnekanti et al, 2002], [Kcman and Fox, 2004]. The use of software components is a key-issue for implementing some of the proposed techniques for improving availability, like micro-reboots [Candea et al, 2003] and path-based failure and evolution management [Chen at al 2004].

Structuring a system in small loosely coupled modules promotes:

1. Increase of control over the development complexity, following the "small is beautiful" lemma [Zisman, 2009];
2. Increase of control over defect detection, as they will tend to be in well isolated points and due to a limited number of factors;
3. Increase the chance of complete recovery in case of failures, due to the possibility (in many cases) of substitution of the defective components for new instances;
4. Increase of software reuse, which promotes the maturity of many modules reducing the failure rates.
5. It is important to notice that the concept of module, in many cases, may be confused with the software component concept – often, software components can be understood as super-modules composed of modules or other components.

2.3.3.
Design by contract and Design for Testability

According to [Payne et all, 1997], early adoption of some activities in the development lifecycle of a system is able to increase its testability. This practice is known as design-for-testability (or DFT), and it is well-known by the hardware development community [Payne et all, 1997]. It is based on the creation of embedded tests and measurement of pre-established parameters in order to facilitate the defect identification and correction in development time. DFT is usually used in the development of complex hardware components in order to guarantee their correct operation before the serial production phase, as the cost of a defect correction in this phase would be prohibitive.

Design by Contract (DBC) [Meyer, 1992] follows the DFT concept. A contract is a formal specification of the state regarding items of the interface of an artifact. It refers to data, functions and methods that are visible in the interface of a class, component, agent or service. Abstract concepts of the artifact may also be used – for example, a stack may be empty or not, so a contract may use expressions that consider these two states. However contracts should not involve attributes and properties that are encapsulated in the artifact. If such items must be made visible they in fact correspond to interface items and, hence, should be described there.

Assertions, on the other hand, are used to specify the internal state of an artifact. They may consider private attributes and methods. Assertions may be:

1. Invariants that define consistency conditions for a state of one or more interdependent objects;
2. Pre-conditions, that define conditions to be satisfied before activating a method; and
3. Post-conditions, that precisely define what a method is supposed to do, i.e., the conditions to be satisfied at the end of a method execution, taking into consideration the different ways of terminating or suspending it.

Assertions and contracts may also establish relations involving elapsed time and other measurable properties such as number of handled transactions and access frequencies of given attributes or methods.

Since assertions may involve encapsulated items as well as items that are visible in the interface we may conclude that contracts are a special case of assertions. We will use the term assertion to denote both contracts and assertions, except when considering explicitly contracts. Keeping up with already established terminology, we will use the term Design By Contract (DBC) to denote the specification of contracts (interface properties) as well as of assertions (encapsulated properties).

Assertions can be checked at runtime, by means of specific code designed to check them. Such code is called executable assertion or executable contract. The use of assertions embedded in code is a common practice for already a long time. However, functions such as "assert" usually involve simple conditional expressions. But when verifying the correctness of data structures or of data collections frequently the code is far more elaborate often requiring quantifiers.

The use of executable assertions may increase the effort spent in modeling and coding phases, but it reduces significantly the test and acceptance phases as it promotes a kind of "self-evaluation and test at runtime" [Staa, 2000; Gotlieb and Botella, 2003] considering the components in which they are applied. There is also a huge potential to reduce the effort spent in development due to incorrect, incomplete or inconsistent specifications, since such mistakes will become apparent when trying to write assertions. Another positive aspect when using design by contract is the increase of the efficiency in failure detection and defect removal at production time as well as in beta versions of the software. This is due to:

1. It is highly probable that the location of the defect is limited to the lines of code executed from the point where assertions were checked last up to the point where an assertion is found to be broken (i.e. the failure is observed);
2. The early detection of a failure by means of an assertion allows gathering useful information to help finding the defect.
3. During production time, defects are prevented from being introduced due to the more formal approach inherent to design by contract. Thus, the software will contain fewer defects to start with.

When developing a system using DBC, it is up to the client (client programmer, calling method, etc.) to assure that all pre-conditions are satisfied. It

is up to the developer of the method to assure that all invariants and post-conditions will be satisfied. However, when designing for testability, one must assume that developers may not have assured these conditions, so it is necessary to add code to verify the contract even in parts where it could be assumed that it is valid. Usually, instead of assuming that all contracts are obeyed, it is assumed that they might not have been obeyed. This implies adding verification code at the beginning and immediately after the execution of a method or a function. This could be achieved with an instrumentation wrapper as shown in Figure 1. Since verifying an invariant of a large data structure might prove to be to costly, verifying the invariant just at the local context of the call might be sufficient in most cases. For example, instead of verifying a whole list, one might verify just the node of the list that will be handled by the specific call.

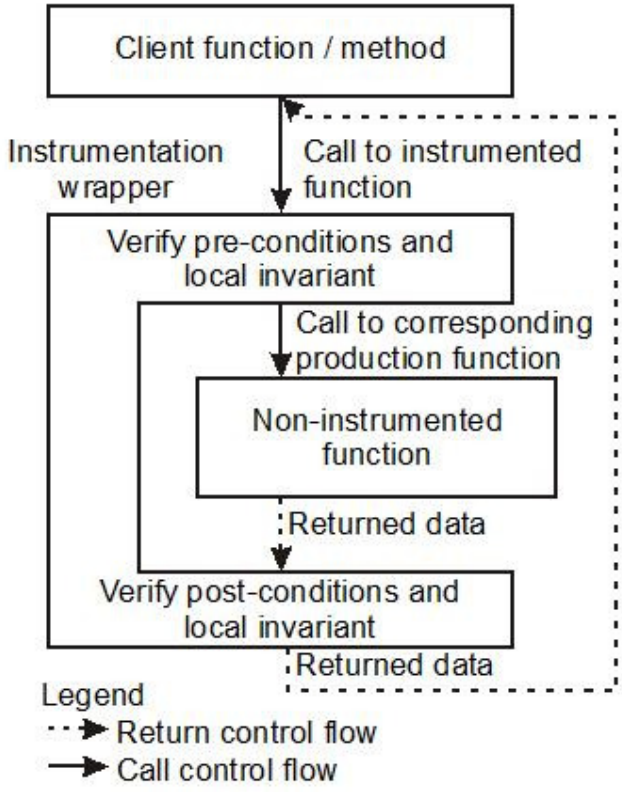

Figure 1: An instrumentation wrapper encapsulating the verification

Executable assertions may be kept in the release version as an error detection tool. Executable contracts may be kept in the release version for components that shall interact with unreliable artifacts.

2.3.4. Mock Components

Mock objects are a test pattern proposed in [Mackinnon et al, 2001] and [Hunt and Thomas, 2003] where an object is replaced by an imitation (the mock object) that simulates the behavior of the object during a test. In our work, we extended this concept to what we call *mock components*. These are groups of modules and classes that can be replaced by imitations in order to ease the test of other parts of a system. A mock component can be an entire subsystem, a component, a software agent or, in the simpler case, a single class. A mock component must have the same interface of the replaced element.

Mock components could be used when one or more of the following conditions occur [Hunt and Thomas, 2003]:

1. The real element has a non-deterministic behavior;
2. The real element is hard to configure;
3. The real element may have abnormal behavior that is hard to reproduce (for example, a connection error); or the test needs to simulate some form of malfunction
4. The real element is too slow;
5. The real element does not yet exist;
6. The test needs to perform measurements such as to gather information about the frequency of use of the real element (for example, how many times a service is being used).

The most frequent use is to obtain information about how the element is being used, measuring:

1. Which services were executed during the test;
2. How may times a service has been called;
3. In which order a set of services has been executed;

4. What values were returned or passed as arguments.

In this work mock components were used to simulate parts of the system that were being developed by different teams, or parts that would be developed in the future. Mocks were also heavily used to simulate abnormal conditions, in order to check if the system was really robust enough to recover from them. This is a possible strategy when applying the defect based test technique proposed in [Morell, 1990], as one can verify experimentally that a set of possible failures is effectively under control.

There are some Java APIs that enable the Mock Objects mechanism like Easy Mock [EasyMock, 2007], JMock [JMock, 2007] and Mock Maker [MockMacker, 2007].

At first sight, a mock component may look like a simple stub, but there are some characteristics that make it more than just a stub [Fowler, 2007]. While a stub is usually used to return specific data when a service is invoked with specific arguments or to execute small tests, the mock is used to gather information about the execution of an element. In spite of this difference, tools used to generate mocks are usually efficient for creating stubs.

It is important to state that the use of mock components is limited to development time only.

2.3.5. Formal Methods

Another interesting area of research dedicated to developing robust software is using formal methods to specify and develop systems [Sobel, 2002] [Gerhart et al, 1994] [Hall, 1990].

[Agerholm and Larsen, 1998] propose the use of “lightweight formal methods”. The authors use the term “light” or “lightweight” in the sense of “less-than-completely-formal” or “partial”, where the methods can be used to perform partial analysis on partial specifications, without a commitment to developing and baselining complete, consistent formal specifications. [Fitzgerald and Larsen, 1995] report some similar results when arguing that “proof is not often a high priority. Formal or rigorous reasoning were not felt to be cost-effective”.

Despite of the good results that have been reported in some experiments, some even in industry [Agerholm and Larsen, 1998] [Fitzgerald and Larsen, 1995] [Hall, 1990], the resistance to adoption of formal methods still remains due to the myth of its complexity and extra costs [Holloway, 1997] [Hall, 1990].

2.3.6. Patterns

A pattern is a general reusable solution to a commonly occurring problem in software. Design patterns deal specifically with problems at the level of software design. Other kinds of patterns, such as architectural patterns, describe problems and solutions that have alternative scopes [Gamma et al, 1995].

Patterns can speed up the development process by providing tested, proven development schemata. Effective software design and architecture require considering issues that may not become visible until later in the implementation. Reusing patterns helps to prevent subtle issues that can cause major problems, and it also improves code readability for coders and architects who are familiar with the patterns.

In this work, some architectural and design patterns were fundamental for achieving the desired level of reliability. Some of them are specific for detecting failures; some are specific for development; some of them are destined to failure handling. We discuss this below.

2.3.6.1. Design Patterns

In this chapter, we discuss some existing design patterns that had proven to be useful when applied to our experiments with Recovery Oriented Software development.

2.3.6.1.1. Adapter

The adapter pattern, often referred to as the wrapper pattern or simply a wrapper, was originally thought to translate one interface for a class into a compatible interface [Gamma et al, 1995]. However, this concept can be extended to components. An adapter allows components to work together that normally

could not because of incompatible interfaces, by providing its interface to clients whilst using the original interface. The adapter translates calls to its interface into calls to the original interface, and the amount of code necessary to do this is typically small.

Another use for the adapter pattern is to decouple the client (user of the services provided by a component) from the server (provider of the services) by defining a unique interface used by the client, which should be implemented by any server that is to be used. It is, during development time, suitable for creating mock elements that are to replace internal parts of a software.

2.3.6.1.2. Decorator

The decorator pattern can be used to make it possible to extend (decorate) the functionality of a class at runtime [Gamma et al, 1995]. This works by adding a new decorator class that wraps the original class. This wrapping is typically achieved by passing the original object as a parameter to the constructor of the decorator when it is created. The decorator implements the new functionality, but for functionality that is not new, the original (wrapped) class is used. The decorating class must have the same interface as the original class.

Decorators can be used to write pre and post conditions for methods. The idea is to separate the method implementation, which contains the business logic, from the pre and post conditions. This allows for running the component with, or without, the verifications, (which may significantly affect runtime performance) and may enhance the overall quality of the design and code as it separates two independent concerns.

2.3.6.1.3. Abstract Factory and Factory Method

A factory is the location in the code at which objects are constructed. The intent in employing factory patterns is to insulate the creation of objects from their usage. This allows for new derived types to be introduced with no change to the code that uses the base class [Gamma et al, 1995].

Factory patterns must be combined with adapters (mock elements) and decorators (pre and post conditions) to reduce as much as possible the need of writing new code.

2.3.6.1.4. Singleton

The singleton pattern is used to restrict instantiation of a class to a specific number of objects [Gamma et al, 1995].

Singletons must be used to restrict some classes to have a unique instance like loggers or failure handlers.

2.3.6.1.5. Observer

The observer pattern (sometimes known as publish/subscribe) allows an object to maintain a list of its dependents, and notifies them automatically of any state changes, usually by calling one of their methods [Gamma et al, 1995]. However, this concept can be extended to allow for the notification not only of state changes, but also for any relevant event that might be of interest of the dependents, like important method invocations, current state (and not only state changes) or identified failures at runtime.

Observers can be used to write failure detectors and failure handlers. For example, imagine a situation where a method that accesses a specific database table is invoked. This method may have a *method execution observer*, which concerns are related to guaranteeing that the database is consistent. If any database inconsistency is detected, some cleanup must be executed and, if the problem may be fixed, the execution can continue normally, without any knowledge of the called method. However, if the problem cannot be properly fixed, it may be a good idea to terminate the feature execution. This can be achieved by raising an exception. If the problem is too serious, software termination can be also considered as a plausible solution, in order to avoid further damage. The piece of code below illustrates this situation.

```
-- Interface for the observer
public interface Observer
{
   void reportDatabaseAccess();
}
--- Class that uses the database
private void reportDatabaseAccess()
{
   for ( Iterator<Observer> it = observers.iterator();
   it.hasNext(); ) {
      it.next().reportDatabaseAccess();
   }
}

public void addUser( User user ) throws InvalidDBStateException
{
   reportDatabaseAccess();
   // add user
}

public void removeUser( User user ) throws InvalidDBStateException
{
   reportDatabaseAccess();
   // remove user
}

public void updateUser( User user ) throws InvalidDBStateException
{
   reportDatabaseAccess();
   // update user
}
```

It is important to notice that failure detectors and failure handlers written as observers completely separate business logic from failure detection/handling concerns, and may also promote reuse of code – consider again the example of the database access. This code can be used throughout the software, in any place where a database access occurs. If the database is too large, it is also possible to write specific table verifications, which shall be invoked whenever necessary.

Observers can also be very useful for logger implementations.

2.3.6.1.6. Memento

The memento pattern provides the ability to restore an object to a previous state [Gamma et al, 1995]. Even though it is primarily used for undo purposes, mementos may be very useful for recovery code of operations that change the state of many objects.

Imagine data acquisition software that has to perform the following operations:

- Acquire data from a set of sensors of different types: ultrasound levels, magnetic fields, geometric profiles, temperature and pressure levels;
- Give these data to handlers that shall perform operations on these values. A handler may perform operations with more than one type of value, and should also define its internal state according to the values received.

Consider that these values are only meaningful if used together – this means that, if a failure is detected in one handler, all data should be discarded. A way to implement this is using the memento pattern for the handlers.

Before initiating the refresh process, a memento for each handler should be created. If any inconsistency is detected, for example, by raising an exception, the previous state must be restored. The following piece of code illustrates the proposed solution:

```
public void newSample( Sample sample ) throws
InvalidSampleException {
   Map<Handler, Memento> m = new HashMap<Handler, Memento>();
   try {
      for ( Iterator<Handler> it = handlers().iterator();
 it.nasHext() ; ) {
         Handler h = it.next();
         m.add( h, h.getMemento() );
         h.newSample( sample );
      }
   } catch ( InvalidSampleException ex ) {
      for ( Iterator it = m.keys().iterator(); it.nasHext() ; ) {
         Handler h = it.next();
         h.setMemento( m.get( h ));
```

```
        }
        throw ex;
    }
}
```

This guarantees that not only a valid state for the handlers shall be maintained, but also that the invalid sample event will be reported (by raising an exception).

2.3.6.2. Architectural Patterns

In this chapter, we discuss some new and existing architectural patterns that had proven to be useful when applied to our experiments with Recovery Oriented Software development.

2.3.6.2.1. Watchdog

A watchdog is a component that has its own thread of execution and constantly checks for specific conditions, which indicate that the overall system is working properly. If any invalid condition is detected, actions in order to restore proper system execution must be carried out. For example, restarting affected components, or even restarting the whole system. A watchdog must never hang due to errors, and must be simple enough to be reliable.

The general structure for a watchdog is as follows:

```
public class Watchdog extends Thread
{
   public void run()
   {
      while ( systemRunning() ) {
         if ( !check_conditions_ok() ) {
            fix();
         }
         sleep_for_some_time();
      }
   }
}
```

It is important to state that the `fix`, `systemRunning` and `check_conditions_ok` methods must be simple enough in order to guarantee that the watchdog itself is not going to hang. Another important issue is that a software watchdog heavily relies on operation systems reliability to properly work.

A watchdog may have to check structures that are under constant evolution. This implies that some sort of synchronization between the watchdog and methods that change the structure must me implemented. This may impose a challenge: if the structure is big, simply locking it for reading or writing may have significant impact on the overall performance of the system. A way to avoid this is to break the watchdog into small pieces, each one responsible for checking a specific part of the structure. Methods that change the structure must them inform the watchdog whenever they are executing, and this can be done by setting a "dirty bit". This requirement must be well documented so that developers follow it.

A watchdog may be implemented completely within the software, but also with some part outside it by means of an external hardware. A very simple watchdog can be built with a hardware based on a 74423 TTL monostable [TTL, 2008], in such a way that:

1) The monostable signal pin is connected to a processor reset pin; and

2) The system constantly triggers the monostable clock.

If the software hangs, the monostable will not be triggered and after some (pre-defined) time, the processor will be reset. Note that this hardware implementation replaces the software watchdog component.

Simple watchdogs can be easily used with stateless systems, as this kind of system can be restarted without the need of any further state restoration.

An important issue to address is that watchdogs must be as simple as possible. This will reduce the chances of faults due to code complexity.

2.3.6.2.2. Secure state change

The secure state change guarantees that after a series of complex (and maybe tied) operations, either a valid state is achieved, or the previous valid state

is restored. If any of the operations fails, the complete process is rolled back to the previous valid state.

The general structure for a secure state change is as follows:

```
--- Interface for the handler.
public interface Handler
{
   public Memento getMemento();
   public void setMemento();
}
--- Method that encapsulates some complex operations.
--- Imagine that there are n handlers to perform complex
operations.
public void complexOperations( Parameter parameter )
      throws InvalidParameterException {
   Map<Handler, Memento> m = new HashMap<Handler, Memento>();
   try {
         m.add( handler_1, handler_1.getMemento() );
         h_1.doComplexOperation(parameter);
         m.add( handler_2, handler_2.getMemento() );
         h_2.doAnotherComplexOperation(parameter);
         m.add( handler_3, handler_3.getMemento() );
         h_3.doAThirdComplexOperation(parameter);
         (…)
         m.add( handler_n, handler_n.getMemento() );
         h_n.doLastComplexOperation(parameter);
      }
   } catch (InvalidParameterException ex) {
      for ( Iterator it = m.keys().iterator(); it.nasHext() ; ) {
         Handler h = it.next();
         h.setMemento( m.get( h ));
      }
      throw ex;
   }
}
```

This guarantees that not only a valid state for the system is maintained, but also that the invalid parameters will be reported (by raising an exception). Note that this architectural pattern is instantiated in the Design Patterns section, illustrating the use of Memento pattern.

2.3.6.2.3. Dirty row flag

The idea with the dirty row flag is to mark bad records in a database so that they will not be used under normal software use.

The general structure for this is to add a specific column in tables to mark the bad records. Every query executed to retrieve data from these tables must then include the state of this column, so that no broken information is retrieved during normal runtime operation (but allowing further retrieve for other purposes, as discussed below). This can be easily done either manually, or with the aid of some relational-object mapping framework like Hibernate [Hibernate, 2008].

The dirty columns must be marked by a database integrity verifier. They can be further examined for two main purposes:

Restore information, avoiding loss of work; and

Identify and correct software bugs that led to the inconsistent state.

2.3.6.2.4. Application restarter

This strategy is proposed in [Fox, 2002], but without the status of an architectural pattern. It consists in restarting software from time to time, as an efficient way to improve availability. This is an interesting claim if we understand it as a way to say that "it is very hard, with the tools and techniques available at the present time, to build a system that executes forever". Restarting a system can be a simple and efficient way to free leaked resources, especially those not allocated by application code. Consider a system written in Java that runs on a virtual machine. The virtual machine may fail to release resources already released by an application's code. This is also true if a 3rd party library is used – no one can guarantee that it is free of leakages.

2.3.6.2.5. Inverse function

"Inverse functions" can be used to double check results. For example, consider a case where a color chart for thirty channels acquired from a specific

inspection tool must be drawn in a window, where each row of the window refers to a specific channel, and each group of columns represent a time position. Consider that a click on a window must give the value of the channel drawn on the clicked position. A way to implement this is to create a pair of functions to map a pixel to a channel (pixelToChannel) used for the click process, and a channel to its pixel (channelToPixel) used for channel drawing. It is easy to notice that, for a given channel c:

```
pixelToChannel(channelToPixel(c)) = c
```

And that, for a given pixel p:

```
channelToPixel(pixelToChannel(p)) = p
```

This condition could be used to generate code that, in both functions, invokes the inverse one in order to verify consistency. A way to avoid an infinite loop would be to create two auxiliary functions, with private scope, that calculate the values, and two public functions, that check for the conditions:

```
int channelToPixelInternal( int c ) {
  // calculate the pixel p
   return p;
}

int channelToPixel( int c ) {
   int p = channelToPixelInternal( c );
   int calcC = pixelToChannelInternal( p );
   if ( calcC != c ) {
    // error!
   }
   return p;
}

int pixelToChannelInternal( int p ) {
  // calculate the channel c
   return c;
}

int pixelToChannel( int p ) {
   int c = pixelToChannelInternal( p );
   int calcP = channelToPixelInternal( c );
```

```
   if ( p != calcP ) {
      // error!
   }
   return c;
}
```

Sometimes, there is no need for the inverse function, from the functional requirements point of view – this is true when one side for the conversion is not required. Even on such occasions, it might be interesting to write the function, for consistency check.